Theoretical Computer Science 18 (1982) 1-31
North-Holland Publishing Company

ABSTRACT DATA TYPES AND ALGEBRAIC SEMANTICS OF PROGRAMMING LANGUAGES

C. PAIR
Centre de Recherche en Informatique de Nancy, 54037 Nancy Cedex, France

Communicated by M. Nivat
Received March 1980
Revised October 1980

1. Introduction

The origin of this study is the research of a formal description of compilers allowing proofs of correctness and automatic generation [9].

If the generation of syntactic analyzers from a grammar has now reached an industrial level [4], writing the other components of a compiler is still generally an ad-hoc work. The reason is probably the absence of well suited tools for defining semantics. However several authors have proposed to use a formal semantics to implement a language; for example, P.D. Mosses [20], N. Jones and A. Schmidt [16] use a denotational semantics of the source language, the target language being fixed.

Following [19] and [3], a compiler is correct if the diagram of Fig. 1 commutes: L_S is the source language, L_T the target language, M_S and M_T the associated sets of meanings, *sem* the semantic mapping, *compil* the compilation mapping, M_S and M_T are connected by a relation *repr* of representation.

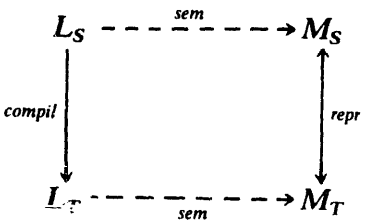

Fig. 1.

A possible solution would be to build actually a compiler with three steps: $L_S \to M_S \to M_T \to L_T$. However, the meanings, defined, e.g., by a denotational semantics, are often complex objects. We prefer to consider intermediate sets of more tractable 'semantical descriptions', D_S for the source language, D_T for the target language, and thus define the compiler according to Fig. 2:

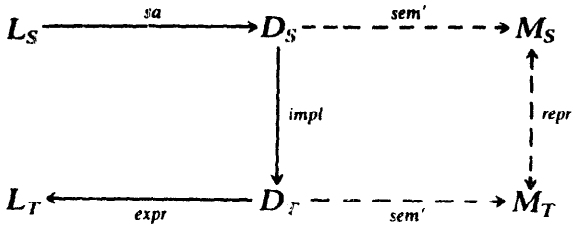

Fig. 2.

- the first step, *sa*, can be considered as a syntactical abstraction [18]; it depends on the source language only;
- the second step, *impl*, depends on the implementation choices;
- the third step, *expr*, is an expression mapping, inverse of a syntactical abstraction; it depends on the target language only.

The nature of the 'semantical descriptions' remains to be chosen. We propose to use terms of algebraic abstract data types [1, 12]. Therefore, for each language we define an abstract type by operations and equational axioms; semantical descriptions will be certain terms of this abstract type (cf. Fig. 3, where T_S and T_T are the sets of terms of the source type and the object type, respectively).

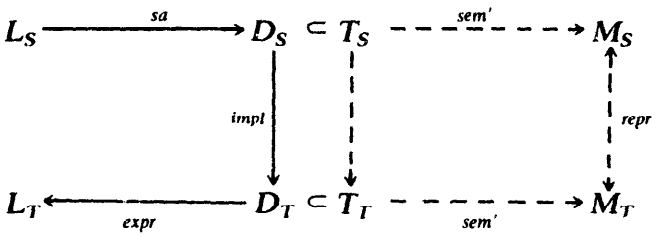

Fig. 3.

This choice, also proposed by [21] and [5], presents several advantages:
- from a pragmatic point of view, the data structure processed by the language has to be described, and an abstract data type is a tool for that description;
- from a more theoretical point of view, the semantics of an abstract type (*sem'* mapping) can be canonically described; a possible choice is 'initial algebra semantics', where M_S and M_T are the initial algebras of the source and target types; but it is not the only possible choice and it can be criticized: other algebras can be chosen for M_S and M_T.
- *sem'* is a morphism from the term algebra T_S (resp. T_T) into the algebra M_S (resp. M_T); *repr* and *impl* are also morphisms: in particular, this means that *impl* is completely defined by the images of the source type operations, these images verifying the source axioms; for initial algebra semantics, morphisms *sem'*, *impl* and *repr* are unique and the diagram commutes.

This leads to define the semantics of a programming language by:
- an algebraic abstract data type;
- a syntactical abstraction mapping (which can, for example, be defined as a morphism on syntactical trees).

Using an algebraic abstract data type for defining semantics has also been done, and its advantages justified, by [29] and [6]. In fact, several semantics are thus defined, according to the choice of the algebra of meanings: a possible choice is the initial algebra; another one would be a terminal algebra. This kind of definition of the semantics can be called *algebraic* since it uses algebras and morphisms, but also in a sense close of [22], for the meaning of a program is then a set of terms, i.e. trees, equivalent for some congruence: with each phrase is associated, by syntactical abstraction, a term of the type, and therefore its class for a congruence, for example generated by the axioms of the type if the chosen semantics is that of initial algebra. Thus, the presence of axioms prevents of having to consider infinite trees like [1] or [22]. The study of models of abstract types by algebraic methods [24, 5] avoids any recourse to fixpoint theory [26]. Here we do not build an 'algebraic denotational semantics', i.e. a formalization of denotational semantics by an abstract type [20]; the abstraction will start from the language itself and not from a denotational semantics: we think that this approach leads to simpler descriptions.

The aim of the paper is to make this semantics precise; it describes the class of abstract types adapted to deterministic algorithmic languages, in particular without parallelism. Indeed, with all these languages correspond similar types; this can make the implementation definitions easier and permits to study them in general. This class of abstract types will be defined in Section 3, after being introduced by simple examples, but avoiding fixing too much the considered language which is of no much importance. The treatment of more complex constructs will be presented afterwards (Sections 4, 5 and 6) to show that the method is general and can be favourably compared to others for simplicity. In the Appendix a complete example, relatively simple but containing some of the difficulties present in programming languages, is treated. In Section 7 are established general results on the considered class of abstract types: in particular, are proved the consistency of the type and the existence of models, necessary to actually define a semantics; more precisely, these models extend given models of the primitive types (integers, booleans, ...) on which the built type relies. These results will allow us to come back to compilation (Section 8).

[10] gives another presentation, using abstract types only for the instantaneous aspects of semantics, the dynamic aspects being presented at another level, less formal, by transformations between abstract types. It is an intermediate point of view between that presented here and a previous formalization of data structures by equational formal systems [23], applied by [8] to the semantics of programming languages.

2. Introduction of the abstract type associated to a programming language

2.1. Signature of the type

To describe a programming language, we have to consider:

- the values to be processed: integers, booleans, ...,

- the phrases of the language: expressions, statements, ...

Thus, among the *sorts* of the type will be found:

- sorts for the *values*, as ***Int***, ***Bool***, . . . ,
- sorts for *expressions* of the corresponding types, which will be noted *Int*, *Bool*, . . . ;

a sort ***Stm*** for the *statements*.

Examples of *terms* of sort *Int* can be:

$$mul(nb(3), nb(4)), \qquad plus(val(id(a)), val(id(b)))$$

corresponding to the concrete syntax:

$$3 * 4 \qquad\qquad a + b$$

An example of a term of sort ***Stm*** can be:

$$assign(id(a), plus(val(id(a)), val(id(b)))).$$

The *operations mul* and *plus* are constructors of the sort *Int*:

$$mul : Int \times Int \to Int, \qquad plus : Int \times Int \to Int.$$

They are associated with similar operations of the value sort **Int**:

$$\boldsymbol{mul : Int \times Int \to Int}, \qquad \boldsymbol{plus : Int \times Int \to Int}.$$

An identifier is also a kind of expression: let us call *Ident* the corresponding sort; it is built from a character string of the program (sort *Text*):

$$id : Text \to Ident.$$

Similarly,

$$nb : Text \to Int.$$

val has the functionality:

$$val : Ident \to Int \quad \text{and} \quad assign : Ident \times Int \to \boldsymbol{Stm}.$$

assign is a constructor of the sort ***Stm***. Among the other constructors of this sort can be found:

$$nothing : \to \boldsymbol{Stm}, \qquad conc : \boldsymbol{Stm \times Stm \to Stm}$$

allowing to build statements as sequences of statements.

An expression has to be evaluated into a value, but this evaluation depends on a 'state'. To express this fact, we introduce a *sort S of states* and an operation which evaluates expressions according to states:

$$eval : S \times Int \to \boldsymbol{Int}.$$

In fact, *eval* will be a 'generic' operation, i.e. an operation with a multiple functionality; for example, it has also the functionality $S \times Bool \to \boldsymbol{Bool}$. It is an 'external operation', connecting the sorts of expressions and states to the primitive sorts.

A statement transforms a state into a state; this is expressed by an operation:

$$apply : S \times \mathbf{Stm} \rightarrow S.$$

Intuitively, a state can be viewed as constituted of objects (integers, strings, arrays, identifiers, procedures, ...) connected by functions (e.g., *val*): cf. the data of [23]. In this sense, it encompasses the notions of environment and store of denotational semantics [26, 27]. It can also be viewed as a set of theorems, e.g. $val(id(x)) = 3$: cf. the informations of [23] or [9].

2.2. Axioms

Let us now see the *axioms* for *eval*.

To compute, e.g., *eval(s,mul(e1,e2))*, we first have to compute *eval(s,e1)* and *eval(s,e2)*, then to use the operation ***mul*** on integers:

$$eval(s,mul(e1,e2)) = \mathbf{mul}(eval(s,e1),eval(s,e2)).$$

Such an axiom will be also written for *add* and, more generally, for every operation h having a corresponding operation $\boldsymbol{h}$ on values.

For example,

$$\begin{aligned} eval(s,mul(nb(3),nb(4))) &= \mathbf{mul}(eval(s,nb(3)),eval(s,nb(4))) \\ &= \mathbf{mul}(\mathbf{nb}(eval(s,3)),\mathbf{nb}(eval(s,4))) \\ &= \mathbf{mul}(\mathbf{nb}(3),\mathbf{nb}(4)) \end{aligned}$$

with

$$\mathbf{nb} : \mathbf{Text} \rightarrow \mathbf{Int}, \qquad \mathbf{3} : \rightarrow \mathbf{Text}, \qquad \mathbf{4} : \rightarrow \mathbf{Text}.$$

We have to introduce a value sort ***Text*** associated with the expression sort *Text*. Similarly will be introduced ***Ident*** associated with *Ident*, and $\mathbf{id} : \mathbf{Text} \rightarrow \mathbf{Ident}$. Thus, we make a complete distinction between 'syntactic' sorts for the phrases of the language and 'semantic' sorts for their values. It is however possible to interpret two such sorts, e.g. *Text* and ***Text***, or *Ident* and ***Ident***, by the same set; in other cases, it is more advisable to distinguish between these interpretations, interpreting for example ***Ident*** as a set of locations while *Ident* is considered as a set of simple or subscripted variables.

The case of the operation *val* is different: its evaluation actually depends on the state. To compute *eval(s,val(i))*, we first compute *eval(s,i)* in the sort ***Ident***; then, we use an axiom:

$$eval(s,val(i)) = eval(s,\mathbf{val}(eval(s,i))).$$

val is a new operation introduced for this axiom:

$$\mathbf{val} : \mathbf{Ident} \rightarrow Int;$$

*eval(s,**val**(l))* gives the value of the 'location' l in the 'state' s.

Its evaluation has to be done by an induction on *s*. To express it, we shall introduce operations which will generate statements and then build states by *apply*. One of them is *sub-**val***: its role is to modify ***val*** for one 'location':

$$\textit{sub-val}: \textit{Ident} \times \textit{Int} \rightarrow \textit{Stm}.$$

Then, we can write the axiom:

$$\textit{eval}(\textit{apply}(s,\textit{sub-val}(i,v)),\textit{val}(j)) \quad = \textbf{if } \textit{eq}(i,j) \textbf{ then } v \textbf{ else } \textit{eval}(s,\textit{val}(j)).$$

Moreover, *sub-**val*** is used to precise *assign* by the axiom:

$$\textit{apply}(s,\textit{assign}(i,e)) = \textit{apply}(s,\textit{sub-val}(\textit{eval}(s,i),\textit{eval}(s,e))).$$

Other axioms can be:

$$\textit{apply}(s,\textit{nothing}) = s,$$

$$\textit{apply}(s,\textit{conc}(\textit{stm1}),\textit{stm2})) = \textit{apply}(\textit{apply}(s,\textit{stm1}),\textit{stm2}).$$

2.3. Syntactical abstraction

The role of the syntactical abstraction *sa* is to transform a phrase of the language, like $a := a + b$, into a term of the abstract type, like

$$\textit{assign}(\textit{id}(a),\textit{plus}(\textit{val}(\textit{id}(a)),\textit{val}(\textit{id}(b))))$$

making manifest the syntax hidden operations. The mapping *sa* has an argument which is a phrase, or more precisely a syntactic tree; it is recursively defined from the grammar. For example:

$$sa[\![S \rightarrow I := E]\!] = \textit{assign}(sa[\![I]\!],sa[\![E]\!]),$$

$$sa[\![E \rightarrow E + P]\!] = \textit{plus}(sa[\![E]\!],sa[\![P]\!]),$$

$$sa[\![P \rightarrow I]\!] = \textit{val}(sa[\![I]\!]),$$

$$sa[\![I \rightarrow a]\!] = \textit{id}(a).$$

The sort *Text* is used to initialize the recursive definition.

In the sequel we omit left parts of rules when no ambiguity arises:

$$sa[\![I := E]\!] = \textit{assign}(sa[\![I]\!],sa[\![E]\!]),$$

$$sa[\![S1;S2]\!] = \textit{conc}(sa[\![S1]\!],sa[\![S2]\!]).$$

This last equation is valid only if the language does not contain go to statements (Section 5).

2.4. Case of a language with procedures

We examine here procedures without parameter nor local variables (see Section 4 for a more general case). A procedure identifier has a 'value', which is a statement:

we call *poss* [1] the operation accessing this value:

$$sa[\![call\ p]\!] = call(id(p))$$

with

$$call : Ident \rightarrow \mathbf{Stm}$$

and the axiom

$$apply(s,call(q)) = apply(s,eval(s,poss(q))).$$

The operation *poss* plays, for procedure identifiers, the role of *val* for integer identifiers. A value is assigned to a procedure identifier by a declaration:

$$sa[\![proc\ p = b]\!] = dcl(id(p),sa[\![b]\!]),$$

$$dcl : Ident \times \mathbf{Stm} \rightarrow \mathbf{Stm}$$

with among the axioms:

$$apply(s,dcl(q,m)) = apply(s,sub\text{-}\mathbf{poss}(eval(s,q),m)),$$

$$eval(conc(s,sub\text{-}\mathbf{poss}(r,m)),\mathbf{poss}(r')) = \mathbf{if}\ \mathbf{eq}(r,r')\ \mathbf{then}\ m\ \mathbf{else}\ eval(s,\mathbf{poss}(r')).$$

sub-poss is a new constructor of the sort ***Stm***, playing for ***poss*** the same role as ***sub-val*** for *val*.

poss(q) is an expression; the corresponding values are statements. We have to introduce an expression sort *Stm* associated with the sort ***Stm*** of statements as value sort:

$$poss : Ident \rightarrow Stm, \quad eval : S \times Stm \rightarrow \mathbf{Stm},$$

$$\mathbf{poss} : \mathbf{Ident} \rightarrow Stm, \quad sub\text{-}\mathbf{poss} : \mathbf{Ident} \times \mathbf{Stm} \rightarrow \mathbf{Stm}.$$

Moreover, it would be more clear to distinguish between integer identifiers and procedure identifiers, introducing a sort *Idproc* and an operation $idp : Text \rightarrow Idproc$. The operations *call*, *dcl*, *poss* would then bear on the sort *Idproc*, and ***poss***, *sub-poss* on the associated sort ***Idproc***.

3. Definition of the abstract type

3.1. Value sorts

For each language are given some *value sorts*, and among them necessarily ***Bool*** and ***Text*** (other examples are ***Int***, ***Ident***).

[1] *poss* is an abbreviation for 'possesses'. The reader could see there an influence of Algol 68 [28]. This influence is real and it could be considered that in some respect the proposed semantics is a formalization of that of [28]. See also [8] which presents similar ideas in another way.

These sorts have operations (e.g., ***plus***, ***mul***, ***nb***, ***id***); in particular two operations

$$eq: V \times V \to Bool,$$

$$if\ then\ else: Bool \times V \times V \to V$$

must exist for some value sort V, when they occur in the axioms for *eval*. Axioms are given for the operations, e.g. $plus(x,y) = plus(y,x)$.

3.2. Expression sorts

Each expression sort V is associated with a value sort V.

To build an expression operations, like *plus*, *mul*, *nb*, *id*, are used, associated with operations on values; other operations, like *val*, are used too, we call them *modifiable operations* (the corresponding values are 'modified' by the statements). The operations generating the expression sorts are therefore:

(a) operations $f: V1 \times \cdots \times Vn \to V$, each of them associated with an operation $f: V1 \times \cdots \times Vn \to V$ on value sorts; they are called *basic operations*.

(b) other operations called *modifiable*; with a modifiable operation $f: V1 \times \cdots \times Vn \to V$ another one is associated, $f: V1 \times \cdots \times Vn \to V$ (which will be used in the axioms for *eval*).

No axiom is given for the operations on expressions.

3.3. The sort Stm of statements

It is also a value sort, which can correspond to an expression sort *Stm*. *Stm* is generated by operations of three classes:

(a) for every modifiable operation $f: V1 \times \cdots \times Vn \to V$, the operation $sub\text{-}f: V1 \times \cdots \times Vn \times V \to Stm$.

(b) some operations depending on the language: their domain is a cartesian product of expressions sorts and ***Stm***; examples are *assign*, *conc*, *nothing*; another one is $cond: Bool \times Stm \times Stm \to Stm$.

(c) $if\ then\ else: Bool \times Stm \times Stm \to Stm$.

3.4. The sort S of states

The abstract type contains a sort S of states, with the following operations:

(a) to generate states:

the initial state $init: \to S$,

$$apply: S \times Stm \to S;$$

thus, a state is given by a term

$$apply(\ldots apply(apply(init,m_1),m_2)\ldots, m_n)$$

where $m_1, m_2, \ldots, m_n$ are statements; thus, it can also be viewed as a computation.

(b) an 'external' operation:

$$eval: S \times V \to \boldsymbol{V}$$

for every pair of an expression sort V and the corresponding value sort $\boldsymbol{V}$.

3.5. Axioms

(a) on operations generating statements:

- for each operation g depending on the language, with n arguments, an unique axiom with as left member $apply(s,g(u_1, \ldots, u_n))$ or $g(u_1, \ldots, u_n)$.

Examples:

$$apply(s,assign(i,e)) = apply(s,sub\text{-}\boldsymbol{val}(eval(s,i),eval(s,e))),$$

$$apply(s,cond(b,m_1,m_2)) = apply(s, \boldsymbol{if}\ eval(s,b)\ \boldsymbol{then}\ m_1\ \boldsymbol{else}\ m_2),$$

$$apply(s,nothing) = s,$$

$$while(b,m) = cond(b,conc(s,while(b,m)),nothing).$$

- $\boldsymbol{if\ true\ then}\ m_1\ \boldsymbol{else}\ m_2 = m_1, \qquad \boldsymbol{if\ false\ then}\ m_1\ \boldsymbol{else}\ m_2 = m_2$

(b) on *eval*:

(1) $eval(s,h(u_1, \ldots, u_n)) = \boldsymbol{h}(eval(s,u_1), \ldots, eval(s,u_n))$

if h is a basic operation, with n arguments ($n \geqslant 0$);

(2) $eval(s,f(u_1, \ldots, u_n)) = eval(s,f(eval(s,u_1), \ldots, eval(s,u_n)));$

(3) $eval(apply(s,sub\text{-}f(v_1, \ldots, v_n, v),f(v'_1, \ldots, v'_n)) =$

$$= \boldsymbol{if} \bigwedge_i \boldsymbol{eq}(v_i, v'_i)\ \boldsymbol{then}\ v\ \boldsymbol{else}\ eval(s,f(v'_1, \ldots, v'_n));$$

(4) $eval(apply(s,sub\text{-}f(v_1, \ldots, v_n, v)),f'(v'_1, \ldots, v'_p)) = eval(s,f'(v'_1, \ldots, v'_p))$

if f and f' are modifiable operations, and f is distinct from f'.

3.6. A possible simplification

It is possible to remove the sort S by replacing a state by the modification which generates it from *init*. In the axioms, *apply* is then replaced by *conc*. In fact, the new type is a representation of the previous one, representing S by ***Stm*** and *apply* by *conc*.

This presentation, slightly more economical, is closer of that of [6]; it has been used in [25]. As it is perhaps less intuitive, less generalizable, e.g. to undeterminism, and makes less clear the hierarchy used below in Section 7, we do not use it here.

4. Application to procedures

We introduce (cf. Section 2.4) associated sorts ***Idproc*** and *Idproc* for procedure identifiers, and associated operations ***idp***: *Text* → ***Idproc*** and *idp*: *Text* → *Idproc*. As we have seen in Section 2.4, an expression set *Stm* is associated with ***Stm***. *Stm* has modifiable operations

$$poss: Idproc \to Stm \quad \text{and} \quad \mathbf{poss}: \mathbf{Idproc} \to Stm.$$

Thus there exists

$$\mathbf{sub\text{-}poss}: \mathbf{Idproc} \times \mathbf{Stm} \to \mathbf{Stm}$$

verifying the axioms; in particular

$$eval(conc(s,\mathbf{sub\text{-}poss}(p,m),\mathbf{poss}(q)) = \mathbf{if}\ \mathbf{eq}(p,q)\ \mathbf{then}\ m\ \mathbf{else}\ eval(s,\mathbf{poss}(q)).$$

We suppose that in the language there exists no operation on procedures; then ***Stm*** has no basic operation. If it would not be the case, it would be necessary to distinguish between ***Stm*** and a value sort of procedures, with two inverse transfer functions between them.

4.1. *Procedures without parameter nor local variable*

This case was studied in Section 2.4:

$$sa[\![call\ p]\!] = call(idp(p)),$$

$$call: Idproc \to \mathbf{Stm};$$

axiom: $apply(s,call(q)) = apply(s,eval(s,poss(q)))$.

The operation *poss* is 'modified' by a declaration:

$$sa[\![proc\ p = body]\!] = dcl(idp(p),sa[\![body]\!]),$$

$$dcl: Idproc \times \mathbf{Stm} \to \mathbf{Stm};$$

axiom: $apply(s,dcl(q,m)) = apply(s,\mathbf{sub\text{-}poss}(eval(s,q),m))$.

Recursivity gives no difficulty in this case: *m*, i.e. *sa[[body]]*, may contain *call(idp(p))* as a subterm.

4.2. *Local variables*

For recursive procedures with local variables, we must make a distinction between identifiers and locations: an identifier can designate several locations, depending on the current procedure, and a location contains a value (here, an integer).

This remark leads us to introduce two value sorts:

- *C* for procedure calls,
- *L* for locations

with two operations:

$$\textbf{\textit{des}}: \textbf{\textit{Ident}} \times \textbf{\textit{C}} \rightarrow \textbf{\textit{L}},$$

$$val: L \rightarrow Int \quad \text{(modifiable operation)}.$$

Then, if i is an identifier (of integer):

$$sa[\![i]\!] = des(id(i),cc)$$

where *cc* is the current procedure call; more precisely, *cc* is a modifiable 0-ary operation, changed for each new call and each return (cf. the environment of denotational semantics):

$$cc: \rightarrow C;$$

C is the expression sort associated with $\textbf{\textit{C}}$.

$\textbf{\textit{C}}$ can be generated by an initial call (of the program), and a constructor of new calls; the new call will depend on the state (i.e. the previous computation); to be sure that it is actually new, we put also as argument the called procedure identifier which will be useful in an axiom below:

$$\textbf{\textit{incall}}: \rightarrow \textbf{\textit{C}},$$

$$\textbf{\textit{newcall}}: \textbf{\textit{S}} \times \textbf{\textit{Idproc}} \rightarrow \textbf{\textit{C}}.$$

The axiom for *call* given in Section 4.1 has then to be changed into:

$$apply(s,call(q)) = apply(apply(apply(s,$$
$$\textbf{\textit{sub-cc}}(\textbf{\textit{newcall}}(s,eval(s,q)))),$$
$$eval(s,poss(q))),$$
$$\textbf{\textit{sub-cc}}(eval(s,cc))).$$

Finally, we can express the rules relative to scopes by two axioms for *des*. The second one uses an operation:

$$\textbf{\textit{scope}}: \textbf{\textit{Ident}} \rightarrow \textbf{\textit{Idproc}};$$

$\textbf{\textit{scope}}(i)$ is the identifier of the smallest procedure containing the declaration of the identifier i.

$$\textbf{\textit{eq}}(\textbf{\textit{des}}(i,c),\textbf{\textit{des}}(j,c)) = \textbf{\textit{eq}}(i,j),$$

$$\textbf{\textit{eq}}(\textbf{\textit{des}}(i,\textbf{\textit{newcall}}(s,q)),\textbf{\textit{des}}(j,eval(s,cc))) = \textbf{\textit{eq}}(i,j) \textbf{ and not } \textbf{\textit{eq}}(\textbf{\textit{scope}}(i),q).$$

A way of introducing $\textbf{\textit{scope}}$ is to put, by the syntactical abstraction, with each identifier the identifier of the procedure where it is declared; therefore *sa* is not

exactly a homomorphism dealing with the syntactic tree of the program, but rather with a tree decorated by attributes [17]. However it is also possible to express this decoration in the type itself.

Let us note that the domain of definition of *des* could be precised by a precondition [13] taking into account the scoping rules (see [10] for the use of preconditions).

4.3. Parameters

Let us now suppose that a procedure has one parameter, called by value. We need an operation to access this parameter from the procedure identifier (supposing all procedure identifiers of a program different):

$$\mathbf{vp}: \mathbf{Idproc} \to \mathbf{Ident}.$$

Then, we have to express that, at a procedure call, the formal parameter takes the value of the actual parameter:

$$sa[\![call\ p(e)]\!] = call1(idp(p), sa[\![e]\!]),$$

$$call1 : Idproc \times Int \to \mathbf{Stm},$$

$$apply(s, call1(q,u)) = apply^4(s,$$

$$sub\text{-}\mathbf{val}(\mathbf{des}(\mathbf{vp}(\mathbf{q}),nc), eval(s,u))$$

$$sub\text{-}\mathbf{cc}(nc),$$

$$eval(s, poss(q)),$$

$$sub\text{-}\mathbf{cc}(eval(s,cc)))$$

where $\mathbf{q} = eval(s,q)$ and $nc = \mathbf{newcall}(s,\mathbf{q})$.

For parameters called by reference, we introduce a sort ***Idpar***, the sort ***Ll*** of the corresponding locations (locations of locations), and operations:

- $\mathbf{desp}: \mathbf{Idpar} \times \mathbf{C} \to \mathbf{Ll}$ similar to ***des***,
- $val: Ll \to L$, modifiable, similar to *val* and giving the location of the corresponding actual parameter.

Then, for a procedure having two parameters, the first called by value and the second by reference:

$$\mathbf{rp}: \mathbf{Idproc} \to \mathbf{Idpar},$$

$$sa[\![call\ p(e,v)]\!] = call2(idp(p), sa[\![e]\!], sa[\![v]\!]),$$

$$apply(s, call2(q,u,l)) = apply^5(s,$$

$$sub\text{-}\mathbf{val}(\mathbf{des}(\mathbf{vp}(\mathbf{q}),nc), eval(s,u)),$$

$$sub\text{-}\mathbf{valp}(\mathbf{desp}(\mathbf{rp}(\mathbf{q}),nc), eval(s,l)),$$

$$sub\text{-}\mathbf{cc}(nc),$$

$$eval(s,poss(q))$$

$$sub\text{-}cc(eval(s,cc)))$$

where $q = eval(s,q)$ and $nc = newcall(s,q)$.

The operations *vp* and *rp* can be introduced by *sa*: for *x*, *y* parameters respectively called by value and by reference,

$$sa[\![x]\!] = vp(scope(x)), \qquad sa[\![y]\!] = rp(scope(y)).$$

For procedure parameters, ***sub-poss*** would be used to pass the actual parameter, but moreover an extra parameter should be passed, the current call, to be reused when calling the parameter.

5. Case of a language with jumps

For a language with go to's, the definition of $sa[\![S1;S2]\!]$ given in Section 2.3 as $conc(sa[\![S1]\!],sa[\![S2]\!])$ is no longer valid. We modify it into

$$sa[\![S1;S2]\!] = cong(sa[\![S1]\!],sa[\![S2]\!]).$$

In order to axiomatize the operation

$$cong: \mathbf{Stm} \times \mathbf{Stm} \to \mathbf{Stm}$$

and jumps, we introduce:
- a value sort ***Label*** generated by the operations:

$$\mathbf{idl}: \mathbf{Text} \to \mathbf{Label}, \qquad \mathbf{seq}: \to Label$$

(the 'label' ***seq*** means 'go on sequentially');
- $\mathbf{eq}: \mathbf{Label} \times \mathbf{Label} \to \mathbf{Bool}$;
- an operation $\mathbf{in}: \mathbf{Stm} \times \mathbf{Label} \to \mathbf{Bool}$ indicating the presence of a label in (the text of) a statement;
- a modifiable operation $\mathbf{next}: \to Label$ giving, at each step, the label where to jump (***seq*** if there is no jump).

Then

$$sa[\![go\ to\ i]\!] = goto(idl(i)),$$

$$goto: Label \to \mathbf{Stm},$$

$$apply(s,goto(l)) = apply(s,sub\text{-}\mathbf{next}(eval(s,l))).$$

Moreover, before executing a labelled statement, it is necessary to 'reset' next to ***seq***:

$$sa[\![i:\ S]\!] = lab(idl(i),sa[\![S]\!]),$$

$$lab: Label \times \mathbf{Stm} \to \mathbf{Stm}$$

$$apply(s,lab(l,m)) = apply(s,conc(sub\text{-}\mathbf{next}(\mathbf{seq}),m)).$$

eq and **in** are easy to axiomatize. For example:

$$\mathbf{eq}(\mathbf{idl}(i),\mathbf{idl}(j)) = \mathbf{eq}(i,j),$$

$$\mathbf{eq}(\mathbf{idl}(i),\mathbf{seq}) = \mathbf{false},$$

$$\mathbf{in}(assign(j,v),l) = \mathbf{false},$$

$$\mathbf{in}(goto(l1),l) = \mathbf{false},$$

$$\mathbf{in}(lab(l1,s),l) = \mathbf{eq}(l1,l)\ \mathbf{or}\ \mathbf{in}(s,l),$$

$$\mathbf{in}(cong(s1,s2),l) = \mathbf{in}(s1,l)\ \mathbf{or}\ \mathbf{in}(s2,l).$$

To simplify the next axioms, we associate with **in** an operation *in* generating boolean expressions:

$$in : \mathbf{Stm} \times Label \rightarrow Bool$$

with the axiom: $eval(s,in(s',l)) = \mathbf{in}(s',eval(s,l))$.

Finally, let us come back to the axiomatization of *cong*:

$$cong(s1,s2) = cond(in(s1,next)\ or\ eq(next,seq),$$
$$cong1(s1,s2),$$
$$cond(in(s2,next),cong2(s1,s2),nothing)).$$

Intuitively, *cong(s1,s2)* executes the statements *s1*, *s2* while the value of *next* is *s1* or in *s2*; *cong1(s1,s2)* or *cong2(s1,s2)* is interpreted like *cong(s1,s2)* but the execution begins in *s1* or in *s2*, respectively. This is expressed by two axioms:

$$cong1(s1,s2) = conc(s1,cond(in(s2,next)\ or\ eq(next,seq),$$
$$cong2(s1,s2),nothing)),$$

$$cong2(s1,s2) = conc(s2,cond(in(s1,next),cong1(s1,s2),nothing)).$$

6. Notions on side effects and functions

Up to now, we have considered that an expression did not cause any modification on the state, i.e. the language has no side effect. Side effects can be caused by functions and we have not considered functions either.

Side effects lead to consider that an expression can cause a modification of state and therefore to give also to *apply* the profile

$$S \times V \rightarrow S$$

for every expression sort *V*.

Moreover, the axioms given for *eval* in Section 3.3 have to be changed. For example:

$$eval(s,h(u1,u2)) = \mathbf{h}(eval(s,u1),eval(apply(s,u1),u2))$$

if h is a basic binary operation;

$$eval(s,f(u)) = eval(apply(s,u),f(eval(s,u)))$$

if f is a modifiable unary operation.

It is also necessary to take side effects into account to axiomatize statements. For example:

$$apply(s,assign(i,e)) = apply(s,sub\text{-}\mathbf{val}(eval(s,i),eval(apply(s,i),e)).$$

For functions, it can be considered that they give a result in a fixed location represented by a modifiable 0-ary operation *res*. Let us suppose that the execution of a function is terminated by that of a return statement indicating the expression the value of which is returned:

$$sa[\![return\ e]\!] = ret(sa[\![e]\!]),$$

$$apply(s,ret(v)) = apply(s,sub\text{-}\mathbf{res}(eval(s,v))).$$

Moreover, for a function call with an integer argument and an integer result:

$$sa[\![p(e)]\!] = fcall(idp(p),sa[\![e]\!]),$$

$$fcall: Idproc \times Int \rightarrow Int$$

$$eval(s,fcall(q,u)) = eval(apply(s,fcall(q,u)),res).$$

apply(s,fcall(q,u)) is axiomatized, according to the peculiarities of the language, in a similar way as in Section 4, taking into account the side effect of the argument.

Parameter functions, and in particular calls by name, could be axiomatized on this basis.

The class of abstract types considered in Section 3 for the languages without side effects can be viewed as a particular case of the class studied here, with the axiom $apply(s,e) = s$ for every expression e.

In the sequel, we study properties of the abstract type in the case where there is not side effect.

7. Study of the type

7.1. Hierarchical construction of the type

Defining the semantics of a programming language consists of extending the semantics, supposed to be known, of primitive objects (integers, booleans, ...) to

the other objects, especially the constructs of the language. This remark leads to give a hierarchical definition of the abstract type associated with the language. It relies on a primitive type, formed with certain value sorts, like ***Int***, ***Bool***, ***Text*** (but not ***C***, ***N*** which can be called *auxiliary sorts*): among the primitive sorts are to be found those sorts the interpretation of which is imposed; in this sense, ***Ident*** can be undifferently considered as a primitive sort or an auxiliary sort.

Moreover, the terms of sort S are in the form

$$apply(\ldots apply(apply(init,m_1)m_2)\ldots, m_p)$$

where the m_i are terms of sort ***Stm***:

$$m_i = sub\text{-}f(v_1, \ldots, v_n, v) \quad \text{or} \quad m_i = g(u_1, \ldots, u_n).$$

Some terms of sort S express *computations*: those where the second argument of *apply* is always a $sub\text{-}f(v_1, \ldots, v_n, v)$. Computations are generated by *init* and 'hidden' operations *subst-f* defined by:

$$apply(s, sub\text{-}f(v_1, \ldots, v_n, v) = subst\text{-}f(s, v_1, \ldots, v_n, v).$$

A theorem $apply(init,m) = cp$, where cp is a computation, means that the program expressed by m terminates; cp can be viewed as a normal form for $apply(init,m)$. We call *normal type* the type where the terms of sort S are restricted to be computations, and *normal terms* its terms.

The type can then be stratified into five levels.

(a) *Primitive level*: a primitive type is given, with primitive sorts, primitive operations, and axioms on them; ***Bool*** is a primitive sort and it is supposed that the primitive type is consistent (i.e. ***true*** = ***false*** is not a theorem).

(b) *Language level*: new sorts correspond to the phrases of the language:

- expression sorts, with operations of profile $V_1 \times \cdots \times V_n \to V$ where the V_i and V are expression sorts;
- ***Stm***, generated by operations of profile $L_1 \times \cdots \times L_n \to$ ***Stm*** where the L_i are expression sorts or ***Stm***.

A term of expression sort will be simply called an expression and a term of sort ***Stm*** a statement. At this level can be defined the syntactical abstraction *sa*.

(c) *Data structure level*: at this level auxiliary sorts and the sort S of states are introduced.

Auxiliary sorts are defined with *auxiliary operations*, using in their profile primitive sorts, S and auxiliary sorts; axioms can be given on them, where only primitive and auxiliary operations occur. Auxiliary sorts allow to associate with each expression sort V a *value sort* $\boldsymbol{V}$, i.e. a primitive sort, an auxiliary sort or ***Stm***, and with some operations of profile $V_1 \times \cdots \times V_n \to V$ a primitive or auxiliary operation of profile $\boldsymbol{V}_1 \times \cdots \times \boldsymbol{V}_n \to \boldsymbol{V}$ (these operations are called basic operations; other operations generating expressions are called modifiable).

To generate S, ***Stm*** is enriched by operations associated with modifiable operations:

$$sub\text{-}f: \boldsymbol{V}_1 \times \cdots \times \boldsymbol{V}_n \times \boldsymbol{V} \rightarrow \boldsymbol{Stm}.$$

S is generated by $init: \rightarrow S$ and $apply: S \times \boldsymbol{Stm} \rightarrow S$ where the second argument is restricted to be a $sub\text{-}f(v_1, \ldots, v_n)$.[2]

(d) *Evaluation level* (*normal type*): the operation *eval*, with profiles $S \times V \rightarrow \boldsymbol{V}$, connects every expression sort V with its associated value sort $\boldsymbol{V}$. To write its axioms (Section 3.5), for each modifiable operation $f: V_1 \times \cdots \times V_n \rightarrow V$ another one, $f: \boldsymbol{V}_1 \times \cdots \times \boldsymbol{V}_n \rightarrow \boldsymbol{V}$, is introduced.

Remark: a slight generalization causes no difficulty: in the preceding profiles of basic or modifiable operations, the V_i could be not only expression sorts, but also ***Stm*** (an example is the operation *in* of Section 5); the convention to be made is that $\boldsymbol{V}_i$ is ***Stm*** when V_i is ***Stm***; moreover, in the axioms (1), (2) of Section 3.5, if u_i is a statement, $eval(s,u_i)$ has to be replaced by u_i.

(e) *Interpretation level* (*full type*): here, terms of sort S are unrestricted. For every operation $g: L_1 \times \cdots \times L_n \rightarrow \boldsymbol{Stm}$ is introduced one, and only one, axiom of one of the two forms:

(5) $apply(s, g(u_1, \ldots, u_n)) = \phi(s)$ or (6) $g(u_1, \ldots, u_n) = \Gamma$.

We make the hypothesis that the term $\phi(s)$ contains s. Moreover the operation ***if then else*** $: \boldsymbol{Bool} \times \boldsymbol{Stm} \times \boldsymbol{Stm} \rightarrow \boldsymbol{Stm}$ is introduced with the axioms:

(7) ***if true then*** m_1 ***else*** $m_2 = m_1$,

(8) ***if false then*** m_1 ***else*** $m_2 = m_2$.

The hierarchy must be understood in a strict manner: in the axioms of a level, variables are replaced by terms of this level. In particular, in the axioms for *eval* or auxiliary operations, a variable of sort S is replaced by a computation. In the normal type, *eval* can express the evaluation of an expression after a computation. In the full type, it is is possible to express the evaluation of an expression after the execution of a program. The fact that the axioms on *eval* can be applied to computations only means that a nonterminating program cannot yield any (primitive) result.

7.2. The normal type

Here are studied properties of sufficient completeness and consistency [12] of the normal type relatively to the primitive type. First, we give some definitions. The type is said to be *initially completed* when for every term $f(t_1, \ldots, t_n)$ where f is a modifiable operation and the t_i terms of data structure level, is added a unique

[2] It is also possible not to introduce *apply* and *sub-f* at this level, but in place *subst-f*, see before.

axiom

(9) $eval(init,f(t_1, \ldots, t_n)) = t$

where t is a term of data structure level. We say that *the auxiliary sorts are sufficiently complete* (resp. *consistent*) if the type of data structure level is sufficiently complete (resp. consistent) relatively to the primitive type; indeed, the verification of these properties only use auxiliary and primitive operations.

Theorem 1: *If the type is initially completed, for every computation s and every expression e of the normal type, there exists a term v where eval does not occur, such that* $eval(s,e) = v$.

Proof: Let us choose for the terms $eval(s,e)$ a noetherian order, i.e. where every descending chain is finite:

$eval(s',e') < eval(s,e) \Leftrightarrow s'$ is a proper subterm of s or s,s' identical and e' is a proper subterm of e.

An axiom of Section 3.5 is applicable to $eval(s,e)$:
- (1) if e begins with a basic operation;
- (3) or (4) if e begins with a modifiable operation f and $s \neq init$;
- (2), then (3) or (4) if e begins with a modifiable operation f and $s \neq init$.

In each case, every $eval(s',e')$ contained in the obtained term verifies $eval(s',e') < eval(s,e)$. By repeating the transformation, a term will be attained where each *eval* only has *init* as its first argument.

Thus, we have only to prove the theorem for $eval(init,e)$. It is immediate by structural induction on e, using axioms (1), (2) and initial completeness.

Consequence: If the type is initially completed and if the auxiliary sorts are sufficiently complete, then the normal type is sufficiently complete relatively to the primitive type.

Proof: From Theorem 1, the normal type is sufficiently complete relatively to the type of data structure level, which in turn is sufficiently complete relatively to the primitive type.

For the type studied in Section 4, auxiliary sorts are not sufficiently complete; preconditions should be added, ensuring that axioms are sufficient to compute $eq(des(i,c),des(j,c'))$ (see Section 4.2) in all useful cases.

Theorem 2: *If the auxiliary sorts are consistent, the normal type is consistent relatively to the primitive type* (*and therefore relatively to* **Bool**).

Proof: It is sufficient to prove that the normal type is consistent relatively to the type of data structure level.

Let us orient from left to right the axioms (1), (2), (3), (4) of Section 3.5, the other axioms remaining in both directions. We will show the confluence of the obtained rewriting system (see [14] for conditions of confluence). There exists no superposition between the left-hand sides of oriented rules, therefore no critical pair. Moreover these rules are compatible with the congruence generated by the unoriented axioms (i.e., denoting $\equiv$ this congruence and $\rightarrow$ the direct rewriting, if $t \rightarrow t'$ and $t \equiv t_1$, there exists t'_1 such that $t_1 \rightarrow t'_1$ and $t' \equiv t'_1$); indeed, operations generating expressions occur in no unoriented axiom. It results that in the quotient of the algebra of terms by this congruence, the rewriting system is confluent, and it is therefore also confluent for the terms themselves.

Consistency results from confluence: if $t = t'$ is a theorem, t and t' can be rewritten into the same term t''. For terms of data structure level, these rewritings cannot use the axioms on *eval*. Therefore $t = t''$ and $t'' = t'$ are theorems at data structure level, and thus $t = t'$ is too.

The theorem remains true for an initially completed type: the added axioms (9) are also oriented from left to right and there is still no superposition.

For the example of Section 4, the consistency of auxiliary sorts is easily proved by proving confluence of the rewriting system obtained when orienting the axioms on auxiliary operations (Section 4.2).

7.3. The full type

The consistency of the full type relatively to the normal type, therefore to the primitive type, can be proved in the same way.

Proposition 1: *If the auxiliary sorts are consistent, the rewriting system obtained by orienting from left to right the axioms introduced at interpretation level and leaving unoriented the other axioms, is confluent.*

Proof: The oriented axioms are: for every operation g generating statements, one, and only one, rule

(5) $apply(s, g(u_1, \ldots, u_n)) \rightarrow \phi(s)$ or (6) $g(u_1, \ldots, u_n) \rightarrow \Gamma$

and axioms on ***if then else***; to ensure compatibility with the congruence $\equiv$ generated by the unoriented axioms, we replace the rules on ***if then else*** by

(7′) ***if*** b ***then*** s_1 ***else*** $s_2 \rightarrow s_1$ whenever $b \equiv$ ***true***,

(8′) ***if*** b ***then*** s_1 ***else*** $s_2 \rightarrow s_2$ whenever $b \equiv$ ***false***.

This does not change the rewriting relation. Then, there exists no superposition for left-hand sides provided that the normal type is consistent: this is true from the hypotheses (Theorem 2). Confluence results like in Theorem 2.

Theorem 3: *The full type is consistent if (and only if) the auxiliary sorts are consistent.*

Proof: Use confluence like in Theorem 2.

But here the type is not sufficiently complete in the general case. If it is, every statement has a normal form and this means that every program terminates.

As the type is not in general sufficiently complete, the question arises if, when a model is given for the primitive type (integers, booleans, ...) it can be extended into a model of the full type. This question is studied in [24] and [5]: the considered models interprete operations as partial functions; we say that a term t is *reducible* if there exists a term p of the primitive type such that $t=p$ is a theorem; each model of the primitive type can be extended into a model of the full type, supposed consistent, if every subterm of primitive sort in a reducible term is also reducible.

Theorem 4: *If the auxiliary sorts are consistent and sufficiently complete, each model of the primitive type can be extended into a model of the full type.*

Proof: Let us begin by adding axioms to get initial completeness (Section 7.2): we have seen that the type remains consistent. From the consequence of Theorem 1, if a term of primitive sort is not reducible, it is not normal. Let t be a term of primitive sort containing a nonreducible term t' of primitive sort. We will show that every axiom transforms t into a term containing a nonreducible term of primitive sort; therefore, t is nonreducible. It is obvious if the transformation bears on a term disjoint of t' or on a subterm of t'. If it bears on a term containing t', it is necessarily induced by an axiom of the interpretation level, since t' is not normal. As a statement contains no term of primitive sort, the axiom is necessarily $apply(s,g(u_1,\ldots,u_q))=\phi(s)$ where t' is a subterm of s; therefore t' is a subterm of $\phi(s)$.

7.4. Computation part of a state

In the sequel, we study states, i.e. terms of sort S, without variable. To simplify notations, we denote here *apply* by an infixed dot: a state s is then written

$$init.m_1.\cdots.m_p;$$

a subterm $init.m_1.\cdots.m_q$ for $0\leqslant q\leqslant p$ is called a *factor* of s.

We call *computation part* of s, and we denote by *comp(s)*, the greatest factor of s which is a computation.

The possible transformation of a state into a computation is made by using the axioms (5), (6), (7), (8) of the interpretation level to rewrite their left-hand side. These rewriting rules were already studied in Proposition 1: it can be considered that they define an abstract interpreter, or an operational semantics. The rewriting relation generated by them and the (unoriented) axioms of normal type will be

denoted by $\rightarrow^*$, and the direct rewriting of which it is the reflexo-transitive closure by $\rightarrow$.

Proposition 2: *If $s \rightarrow s'$ by a rewriting rule of the interpretation level, then comp(s) is a factor of comp(s').*

Proof. $s = comp(s).m_{q+1}.\cdots.m_p$. A rewriting rule is applied on a non normal factor:

for (5): $comp(s).m_{q+1}.\cdots.m_r \rightarrow \phi(comp(s).\cdots.m_{r-1})$,

In every case, *comp(s)* is a factor of s', therefore a factor of *comp(s')*.

The unoriented rewriting rules do not possess the same property. To extend it, we consider the congruence $\equiv$ generated by the axioms of the normal type. Since these axioms never equal states, it is obvious that the states congruent to $init.m_1.\cdots.m_p$ are the $init.m'_1.\cdots.m'_p$ where $m_i \equiv m'_i$ $(1 \leqslant i \leqslant p)$. Therefore, if $s \equiv s'$, then $comp(s) \equiv comp(s')$.

The relation 'is a factor of' can be extended into a relation compatible with $\equiv$:

$$s < s' \Leftrightarrow \exists\, s_1\ (s \equiv s_1 \text{ and } s_1 \text{ factor of } s').$$

Proposition 3: *$<$ is reflexive, transitive and compatible with $\equiv$.*

Proof: $<$ is the product of two reflexive and transitive relations $\equiv$ and φ: 'is a factor of'. Moreover these two relations commute: indeed, s is a factor of a state congruent to s' iff s is congruent to a factor of s'. Therefore the product is reflexive, transitive (since $<^2 = \equiv\varphi\equiv\varphi = \equiv^2\varphi^2 = \equiv\varphi = <$) and compatible with $\equiv$ (since $<\equiv = \equiv < = \equiv^2\varphi = \equiv\varphi = <$).

Proposition 4: *If $s \equiv s'$, then $comp(s) \equiv comp(s')$. If $s \overset{*}{\rightarrow} s'$, then $comp(s) < comp(s')$.*

Proof: We still know the first part. Moreover, if $s \rightarrow s'$, then $comp(s) < comp(s')$ by Proposition 2 if the rewriting rule is a rule of the interpretation level, and $comp(s) \equiv comp(s')$ otherwise: in both cases, $comp(s) < comp(s')$. Then, the second part results of the transitivity of $<$.

Moreover the relation $<$ passes to the quotient by $\equiv$ and:

Proposition 5: *In the quotient of the set of states by $\equiv$, the relation $<$ is an ordering; the set of lower bounds of a given $\equiv$-class is finite and totally ordered by $<$.*

Proof: $<$ is reflexive and transitive from Proposition 3.

$<$ is antisymmetric: if $s<s'$, the length (number of factors) of s is at most the length of s'; if $s<s'$ and $s'<s$, s and s' have the same length and therefore the factor of s' congruent to s is s' itself: $s \equiv s'$.

The second part of the proposition results from the fact that the set of factors of s is finite and totally ordered.

7.5. Study of the set of states

The congruence generated by the axioms of the full type (stronger than $\equiv$) divides the states into classes where, for two states s and s', $s=s'$ is a theorem. We study these classes for a consistent type (see Theorem 3), considering the computation parts of their states which can be viewed as beginnings of computations of these states.

Let us suppose that $s=s'$ is a theorem. From Proposition 1, s and s' can be rewritten into the same s''. From Proposition 4, $comp(s)<comp(s'')$ and $comp(s')<comp(s'')$. From Proposition 5, $comp(s)<comp(s')$ or $comp(s')<comp(s)$. We have thus proved that the computation parts of a class (or more exactly their classes for the congruence $\equiv$) are totally ordered by $<$.

Two cases then arise for a class:

(a) the computation parts have a maximal element $comp(s_m)$ and therefore they (or more exactly their classes for $\equiv$) are finite in number (Proposition 5);

(b) the computation parts have no maximal element and they (their classes for $\equiv$) form an infinite increasing sequence: an infinite strictly increasing sequence $comp(s_1), \ldots, comp(s_i), \ldots$, where s_i belongs to the class and $comp(s_i)$ is a factor of $comp(s_{i+1})$, is called an *infinite computation* of the class (it can be considered that it defines an 'infinite term').

If a class contains a computation (i.e. a normal form for its states) s_n, then $comp(s_n)$ is identical to s_n and the only axioms applicable for rewriting s_n are the axioms of the normal type. It results that $comp(s_n)$ is maximal: indeed, if $s=s_n$ is a theorem, s and s_n can be rewritten into some s''; therefore $s_n \equiv s''$ and $comp(s)<comp(s_n)$ (Proposition 4). The class is in the case (a).

But, conversely, the existence of a maximal computation part $comp(s_m)$ for a class does not imply that of a normal form: $comp(s_m)$ can be distinct from s_m; in this case, we say that s_m is a *blocked form*. It is easy to find an example: if an axiom is $s.g(u_1, \ldots, u_n)=s.g(u_1, \ldots, u_n)$, then for every computation s, $s.g(u_1, \ldots, u_n)$ is a blocked form. It can also be verified that a blocked form exists for the language studied in the Appendix, because of the statement *while(true,nothing)*. Intuitively, a program leading to a blocked form does not terminate but does not provoke either an infinity of state transitions; it can be said that it leads to 'busy waiting'. Two kinds of non termination thus exist: infinite computation and blocked form.

To sum up:

Theorem 5: *For a consistent type, the (classes for $\equiv$ of the) computation parts of*

the terms of every class of the congruence generated by the axioms are totally ordered by $<$. *Only three kinds of terms can exist*:

(1) *those admitting a normal form* $s_n(comp(s_n)$ *identical to* s_n *and maximal in its class*);

(2) *those leading to an infinite computation* (*the* $comp(s_n)$ *of the class are not bounded*);

(3) *those admitting a blocked form* $comp(s_m)$ ($comp(s_m)$ *distinct from* s_m *and maximal in its class*).

8. Application to compilation

8.1. Implementation of the abstract type

As we said in the introduction, a compiler can be defined by the abstract type associated with the source language and the syntactical abstraction mapping, the abstract type associated with the target language and the expression mapping, and furthermore an implementation of the source type into the target type. On a similar basis is developed at INRIA a compiler writing system [10, 7].

The implementation is defined according to the hierarchical construction:

- a sort of the source type is implemented by a sort of the target type: a primitive (resp. auxiliary) sort V_S by a primitive (resp. primitive or auxiliary) sort V_T, in particular the standard sorts ***Text*** and ***Bool*** by themselves; the corresponding expression sort V_S by V_T; ***Stm*** and S by themselves.
- an operation of the source type is implemented by a composition of operations of the target type, with correspondence of profiles; in particular the standard operations (boolean operations, ***eq***, ***if then else***, *eval*, *apply*, *init*) are implemented by themselves; the primitive operations are implemented by compositions of primitive operations; each *sub-f* is implemented by some k such that $s.k(u_1, \ldots, u_n)$ has a normal form; another operation g generating statements is not implemented by a *sub-f* of the target type.
- the implementation transforms the axioms of the source type into equations which must be theorems of the target type (weaker representations could be considered, using an interpretation of equality [11]).

It should be noted that the presence of the primitive sort ***Bool*** and the representation of ***eq***, ***true***, ***false*** by themselves prevent trivial implementations where all terms of a sort would be represented by equal terms, and thus prevent from imposing too strong conditions to avoid it, e.g. a one to one correspondence between terms [21].

8.2. Translation of states

Let us study the implementation of states in the three cases of Theorem 5. The implementation of s is denoted by $\bar{s}$.

Firstly, we remark that a theorem $s=s'$ of the source type is translated into a theorem $\bar{s}=\bar{s}'$ of the target type, that a term having a normal form is translated into a term having a normal form, that $\overline{comp(s)}$ has a normal form which is a $comp(\tilde{s})$ where $\bar{s}=\tilde{s}$ is a theorem.

(1) If s has a normal form s_n, $s=s_n$ and therefore $\bar{s}=\bar{s}_n$ are theorems: the normal form of $\bar{s}_n$ is the normal form of $\bar{s}$.

(2) Let us suppose that the class of s has an infinite computation $comp(s_1),\ldots,comp(s_i),\ldots$. As $s=s_i$ is a theorem of the source type, $\bar{s}=\bar{s}_i$ is a theorem of the object type. The normal form of $\overline{comp(s_i)}$ is a $comp(\tilde{s}_i)$ where $\tilde{s}=\tilde{s}_i$ is a theorem; moreover $\overline{comp(s_i)}$ is a strict factor of $\overline{comp(s_{i+1})}$ and therefore $comp(\tilde{s}_i)$ is a strict factor of $comp(\tilde{s}_{i+1}):comp(\tilde{s}_1),\ldots,comp(\tilde{s}_i),\ldots$ is thus an infinite computation of the class of $\bar{s}$.

(3) If s has a blocked form, $\bar{s}$ can be in whatever case: normal form, infinite computation, blocked form. For example, for a 0-ary operation g with the axiom $s.g=s.g$, $\bar{g}$ can be arbitrary:
- with $\bar{g}=nothing$ and the axiom $s.nothing=s$, $init.\bar{g}$ has a normal form,
- with the axiom $s.\bar{g}=s.sub\text{-}f(\bar{u},\bar{v}).\bar{g}$, $init.\bar{g}$ leads to an infinite computation,
- with the axiom $s.\bar{g}=s.\bar{g}$, $init.\bar{g}$ has a blocked form.

Theorem 6: *For a consistent source type:*
– if a state s has a normal form s_n, its translation $\bar{s}$ has a normal form, the normal form of the translation $\bar{s}_n$ of s_n;
– if a state s leads to an infinite computation $\{comp(s_i)\}$, its translation $\bar{s}_i$ leads to an infinite computation constituted of the normal forms of the translations $\overline{comp(s_i)}$.

8.3. Application to the correctness of a compiler

Correctness of a compiler can have several meanings, more and more requiring.

(a) A first meaning is that if the source program yields a result, the target program yields the same result: the result of a program p is the value of an expression of primitive sort; it is formalized by a term $eval(init.m,e)$ where m is $sa[\![p]\!]$. Now, if $eval(init.m,e)=a$, where a is a primitive term, is a theorem of the source type, then $eval(init.\bar{m},\bar{e})=\bar{a}$ is a theorem of the target type and $\bar{a}$ is a primitive term of the target type.

(b) It can also be asserted that if the source program terminates ($init.m$ has a normal form s_n), the target program terminates ($init.\bar{m}$ has a normal form $\tilde{s}_n$, that of $\bar{s}_n$). The partial correctness is thus guaranteed.

(c) For the total correctness, Theorem 6 asserts that a source program leading to an infinite computation is translated into a target program leading to an infinite computation and there is even a correspondence between these infinite computations.

But it can also be required:

(d) that a term $apply(init,sa[\![p]\!])$ without normal form is translated into a term without normal form, so that a nonterminating program is translated into a nonter-

minating program; this is true if a term with a blocked form is not translated into a term with a normal form.

(e) that a term *init.sa[[p]]* having a blocked form is translated into a term having a blocked form, and even that these blocked forms correspond by the translation.

The simplest case where these last two conditions are satisfied is when no term *init.sa[[p]]* has a blocked form.

We call a term *semi-normal* if all its subterms of another sort than S are normal. If s is normal, *s.sa[[p]]* is normal.

Theorem 7: *If*

(1) *every state s.m, where s and m are terms of data structure level, can be rewritten into a semi-normal term* $s.m_1.\cdots.m_q$ $(q \geq 1)$ *where* m_1 *is some sub-f(u,v) or a strict subterm of m,*

(2) *the type is initially completed,*

then a semi-normal state cannot admit a blocked form.

Proof: Starting with a semi-normal state having no normal form, we build an infinite sequence of semi-normal rewritings s_i such that the sequence $comp(s_i)$ is not stationary. At each step

$$s_i = comp(s_i).m.\mu_1.\cdots.\mu_p$$

where $m, \mu_1, \ldots, \mu_p$ are statements; $comp(s_i)$ and m are normal; after eliminating *eval* by Theorem 1, hypothese 1 can be used to rewrite $comp(s_i).m$ into a semi-normal term:

$$s_{i+1} = comp(s_i).m_1.\cdots.m_q.\mu_1.\cdots.\mu_p.$$

If m_1 is some *sub-f(u,v)*, $comp(s_{i+1})$ is greater than $comp(s_i)$; if not, $comp(s_{i+1})$ is identical to $comp(s_i)$, but since m_1 is a strict subterm of m, the sequence of the $comp(s_i)$ cannot be stationary.

For the example in the Appendix, the axioms on the operations generating ***Stm*** directly show that hypothese 1 is satisfied, but for three of them:

- $s.cond(b,m_1,m_2) =$ ***if*** $eval(s,b)$ ***then*** m_1 ***else*** m_2: if preconditions or axioms are added so that auxiliary sorts become sufficiently complete, from consequence of Theorem 1 and from the fact that the primitive type is sufficiently complete relatively to ***Bool***, $eval(s,b) =$ ***true*** or $eval(s,b) =$ ***false*** is a theorem: therefore hypothese 1 is satisfied in this case;
- $s.nothing = s$ and the axiom for *while*, leading to a blocked form for *init.while(true,nothing)*.

If the definition is changed by removing *nothing* and replacing the axiom for *while* by

$$while(b,m) = cond(b,conc(m,while(b,m)),assign(n,val(n))),$$

s.while(b,m) is rewritten into

s.if eval(s,b) **then** *conc(m,while(b,m))* **else** *assign(n,val(n))*

and, then into

s.m.while(b,m) or *s.sub-***val***(n,val(n))*

according to the value of *eval(s,b).* Hypothese 1 is verified.

9. Conclusion

We have shown how an algebraic abstract data type can be associated with a programming language to describe its semantics, and we have seen how the principal aspects of deterministic algorithmic languages could be presented in this frame. The extension to parallelism remains to be studied. Moreover, preconditions must be introduced, particularly to deal with errors.

The abstract type takes together into account the phrases of the language and elements allowing to express their meaning, like states, *eval, apply*; this is different from denotational semantics [26] or from [1] where the syntactic and semantic levels are distinguished, connected by an interpretation function. Grouping together these two levels is specially interesting for procedures, for the value possessed by a procedure is a phrase of the language. Moreover a proof on semantics can be expressed in this formal frame if it relies only on the axioms and not on a particular semantics (i.e. algebra) of the abstract type. These advantages go beyond the application to compilation: every proof, every automatic transformation, can find an advantage in a unified formal frame, using only simple notions and not complicated mathematical objects.

Let us remark that, if the type can be relatively complex, the descriptions of phrases by their syntactical abstraction are more simple (level b of the hierarchical construction of Section 7.1, the other levels being used only to write axioms).

For compilation, the given method, both systematic and practical, divides its complexity and allows to prove correction. Verification of axioms can be done following the hierarchical construction of the type. If, obviously, nothing general can be said for the implementation of primitive or auxiliary sorts, a general study seems possible for the implementation of the *sub-f*, leading to a correction algorithm.

The problem of compiler correction (Section 8) can also be connected to the chosen abstract type semantics. Initial algebra semantics ensures commutativity of the diagram of Fig. 3 (Section 1) but does not guarantee total correctness of the compiler, for it expresses nothing on nontermination. Other semantics can be considered [6], defining stronger congruences on terms: we have spoken of models extending a given model of the primitive type (Theorem 4 and [24, 5]); for them, states without normal form are all considered as equivalent; moreover, choosing among them a terminal model comes back to consider as equivalent the expressions always evaluated (by *eval*) into the same value, the states where the expressions

have the same value (by *eval*), the statements having the same effect (by *apply*): cf. denotational semantics. It is also possible to consider as equivalent, among the states without normal form, only those leading to the same infinite computations: cf. continuous initial algebras [1] Thus, commutativity of the diagram can correspond to one idea, or another, of compiler correctness.

Appendix. Description of a language

Recall of standard elements (present for every language)

- Primitive sorts:

Text: operations: ***empty***: $\rightarrow$ ***Text***
adj-c: ***Text*** $\rightarrow$ ***Text*** for every character ***c***
eq: ***Text*** $\times$ ***Text*** $\rightarrow$ ***Bool***

Bool: operations: ***true, false, not, and***, . . .
if then else: ***Bool*** $\times V \times V \rightarrow V$ for every primitive or auxiliary sort V or for $V =$ ***Stm***

The sort ***Int*** with the usual operations ***plus, eq, inf***, and ***integer***: ***Text*** $\rightarrow$ ***Int*** is also considered as standard.

Axioms on the preceding operations are supposed to be known.

- Sort ***Stm***
- Sort S:

operations: *init*: $\rightarrow S$
apply: $S \times$ ***Stm*** $\rightarrow S$ (denoted by an infixed dot)
eval: $S \times V \rightarrow V$ for every expression sort V and its associated value sort V

Axioms (1), (2), (3), (4) on *eval* are given in Section 3.5.

The description is given in four columns (see Table A.1)

(1) Grammar rules.

(2) Definition of the syntactical abstraction *sa*: the syntactical abstraction of the left-hand side of the rule is given using those of the right-hand side, simply designated by the corresponding nonterminal symbol: when a right-hand side contains several occurrences of the same nonterminal, they are subscripted.

(3) Operations with their profile:

- operations on value sorts (i.e. primitive and auxiliary sorts; these sorts are set using boldface characters); they are extended into basic operations on the corresponding expression sorts.
- modifiable operations, generating expressions sorts (not boldface sorts); they are given with primitive, auxiliary sorts or ***Stm*** in their domain, and they extend into other modifiable operations on the corresponding expression sorts.
- operations generating statements.

(4) Axioms and comments.

Table A.1

Grammar	*sa*	Operations
SS → SS; St	*conc(SS,St)*	***conc : Stm × Stm → Stm***
SS →	*nothing*	***nothing : → Stm***
St → begin SS end	*SS*	
St → if B then SS₁ else SS₂ end	*cond(B,SS₁,SS₂)*	***cond : Bool × Stm × Stm → Stm***
St → while B do SS end	*while(B,SS)*	***while : Bool × Stm → Stm***
St → var I	*nothing*	
St → LP := E	*assign(LP,E)*	***assign : L × Int → Stm*** ***val : L → Int***
B → ¬B	*not(B)*	
B → E₁ = E₂	*eq(E₁,E₂)*	
B → E₁ < E₂	*inf(E₁,E₂)*	
E → E + T	*plus(E,T)*	
E → T	*T*	
T → Nb	*integer(Nb)*	
T → LP	*val(LP)*	
LP → I	*des(I,cc)*	***des : Ident × C → N*** ***cc : → C*** ***newcall : S × Idproc → C*** ***scope : Ident → Idproc***
I → IS.P	*id(IS,P)*	***id : Text × Idproc → Ident***
P → IS	*idp(IS)*	***idp : Text → Idproc***
LP → R	*valp(desp(R,cc))*	***valp : Ll → L*** ***desp : Idpar × C → Ll*** ***scopep : Idpar → Idproc***
R → IS.P	*rp(P)*	***rp : Idproc → Idpar***
T → V	*val(des(V,cc))*	
V → IS.P	*vp(P)*	***vp : Idproc → Ident***
St → proc P = (V,R) St	*dcl(P,St)*	***dcl : Idproc × Stm → Stm*** ***pcss : Idproc → Stm***
St → call P(E,LP)	*call(P,E,LP)*	***call : Idproc × Int × L → Stm***

Recapitulation of sorts:
Primitive sorts: ***Text, Bool, Int***
Auxiliary sorts: ***Ident, Idproc, Idpar, C*** (calls), ***L*** (integer locations), ***Ll*** (location locations)
Expression sorts: associated to the preceding sorts, and ***Stm*** associated with ***Stm***
Stm and ***S***

Axioms and comments

$s.conc(m_1,m_2) = s.m_1.m_2$
$s.nothing = s$

$s.cond(b,m_1,m_2) = s.\mathbf{if}\ eval(s,b)\ \mathbf{then}\ m_1\ \mathbf{else}\ m_2$

$while(b,m) = cond(b,conc(m,while(b,m)),nothing)$
variable declarations are not executed, they define scopes (see below)
$s.assign(l,e) = s.\mathbf{sub\text{-}val}(eval(s,l),eval(s,e))$

$\mathbf{eq}(\mathbf{des}(i,\mathbf{newcall}(s,q)),\mathbf{des}(j,eval(s,cc))) = \mathbf{and}(\mathbf{eq}(i,j),\mathbf{not}(\mathbf{eq}(\mathbf{scope}(i),q)))$
$\mathbf{eq}(\mathbf{des}(i,c),\mathbf{des}(j,c)) = \mathbf{eq}(i,j)$

$\mathbf{scope}(\mathbf{id}(i,q)) = q$
a preprocessor binds with each identifier the identifier of the procedure where it is declared, thus suppressing synonymies (procedure identifiers are supposed to be distinct).
$\mathbf{eq}(\mathbf{id}(i,p),\mathbf{id}(j,q)) = \mathbf{and}(\mathbf{eq}(i,j),\mathbf{eq}(p,q))$
$\mathbf{eq}(\mathbf{idp}(p),\mathbf{idp}(q)) = \mathbf{eq}(p,q)$
R: formal parameter called by reference
$\mathbf{eq}(\mathbf{desp}(i,\mathbf{newcall}(s,q)),\mathbf{desp}(j,eval(s,cc))) = \mathbf{and}(\mathbf{eq}(i,j),\mathbf{not}(\mathbf{eq}(\mathbf{scopep}(i),q)))$
$\mathbf{scopep}(\mathbf{rp}(q)) = q$
$\mathbf{eq}(\mathbf{rp}(p),\mathbf{rp}(q)) = \mathbf{eq}(p,q)$
V: formal parameter called by value
$\mathbf{eq}(\mathbf{vp}(p),\mathbf{vp}(q)) = \mathbf{eq}(p,q), \quad \mathbf{eq}(\mathbf{vp}(p),\mathbf{id}(i,q)) = \mathbf{false}$
$s.dcl(q,m) = s.\mathbf{sub\text{-}poss}(eval(s,q),m)$

$s.call(q,u,l) = s.\mathbf{sub\text{-}val}(\mathbf{des}(\mathbf{vp}(q),nc),eval(s,u))$
$.\mathbf{sub\text{-}valp}(\mathbf{desp}(\mathbf{rp}(q),nc),eval(s,l))$
$.\mathbf{sub\text{-}cc}(nc)$
$.eval(s,poss(q))$
$.\mathbf{sub\text{-}cc}(eval(s,cc))$
where $q = eval(s,q)$ and $nc = \mathbf{newcall}(s,q)$

References

[1] J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright, Initial algebra semantics and continuous algebra, *J. ACM* **24** (1977) 68–95.

[2] J.A. Goguen, J.W. Thatcher and E.G. Wagner, An initial algebra approach to the specification, implementation and correctness of abstract data types, in: R. Yeh, Ed., *Current Trends in Programming Methodology 4* (Prentice-Hall, Englewood Cliffs, NJ, 1978) 80–149.

[3] J.W. Thatcher, E.G. Wagner and J.B. Wright, More on advice on structuring compilers and proving them correct, *Proc. 6th Colloquium on Automata, Languages and Programming*, Graz (1979).

[4] A. V. Aho and J.D. Ullmann, *Principles of Compiler Design* (Addison-Wesley, Reading, MA, 1977).

[5] M. Broy, C. Pair and M. Wirsing, A systematic study of models of abstract data types, 81-R-42, Centre de Recherche en Informatique de Nancy (1981), to appear.

[6] M. Broy and M. Wirsing, Programming languages as abstract data types, *Proc. 5th Colloquium Les Arbres en Algèbre et en Programmation*, Lille (1980).

[7] P. Deschamp, Production de compilateurs à partir d'une description sémantique des langages: le système Perluette, Thèse, Institut National Polytechnique, Nancy (1980).

[8] J.P. Finance, Une formalisation de la sémantique des langages de programmation, *RAIRO Informatique Théorique* **10** (1976), No. 8, 3–32 et No. 10, 5–21.

[9] M.C. Gaudel, A formal approach to translator specification, in: B. Gilchrist, Ed., *Information Processing 77* (North-Holland, Amsterdam, 1977).

[10] M.C. Gaudel, Génération et preuve de compilateurs basées sur une sémantique formelle des langages de programmation, Thèse, Institut National Polytechnique, Nancy (1980).

[11] J.V. Guttag, E. Horowitz and D.R. Musser, Abstract data types and software validation, Report ISI/RR-76-49, University of Southern California (1976).

[12] J.V. Guttag and J.H. Horning, The algebraic specification of abstract data types, *Acta Informat.* **10** (1978) 27–52.

[13] J.V. Guttag, Notes on type abstraction, *Proc. Specification of Reliable Software Conference*, Boston (1979) 36–46.

[14] G. Huet, Confluent reductions: abstract properties and applications to term rewriting systems, *18th IEEE Symposium on Foundations of Computer Science* (1977) 30–45.

[15] S. Igarashi, Semantics of Algol-like statements, in: E. Engeler, Ed., *Semantics of Algorithmic Languages*, Lecture Notes in Mathematics **188** (Springer, Berlin, 1972) 117–188.

[16] N.D. Jones and D.A. Schmidt, Compiler generation from denotational semantics, Computer Sciences Department, University of Aarhus (1980).

[17] D. Knuth, Semantics of context-free languages, *Math. Systems Theory* **2** (1968) 127–145.

[18] J. McCarthy, A basis for a mathematical theory of computation, in: P. Braffort and D. Hirschberg, Eds., *Computer Programming and Formal Systems* (North-Holland, Amsterdam, 1963) 33–704.

[19] F.L. Morris, Advice on structuring compilers and proving their correctness, *Proc. ACM Symposium on Principles of Programming Languages* (1973) 144–152.

[20] P. D. Mosses, Mathematical semantics and compiler generation, Ph.D. Thesis, Oxford (1975).

[21] P. D. Mosses, A constructive approach to compiler correctness, Aarhus University (1979).

[22] B. Courcelle and M. Nivat, Algebraic families of interpretations, *17th Symposium on Foundations of Computer Science*, Houston (1976).

[23] C. Pair, Formalization of the notions of data, information and information structure, in: J.W. Klimbie and K.L. Koffemann, Eds., *Data Management Systems* (North-Holland, Amsterdam, 1974) 149–168.

[24] C. Pair, Sur les modèles des types abstraits algébriques, Séminaire LITP et 80-P-052, Centre de Recherche en Informatique de Nancy (1980).

[25] C. Pair, Application of abstract data types to the definition of the semantics of programming languages, Conference on Formalization of Programming Concepts, Peniscola (1981) and Centre de Recherche en Informatique de Nancy 81-P-025.

[26] D. Scott and C. Strachey, Toward a mathematical semantics for computer languages, *Proc. Symposium Computers and Automata*, Polytechnic Institute of Brooklyn (1971), 19–46.

[27] R. Tennent, The denotational semantics of programming languages, *Comm. ACM* **19** (1976) 437–453.

[28] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Report on the algorithmic language ALGOL 68, M.R. 101, Mathematisch Centrum, Amsterdam (1969).

[29] M. Wand, First-order identities as a defining language, T.R. 29, Computer Science Department, Indiana University (1979).